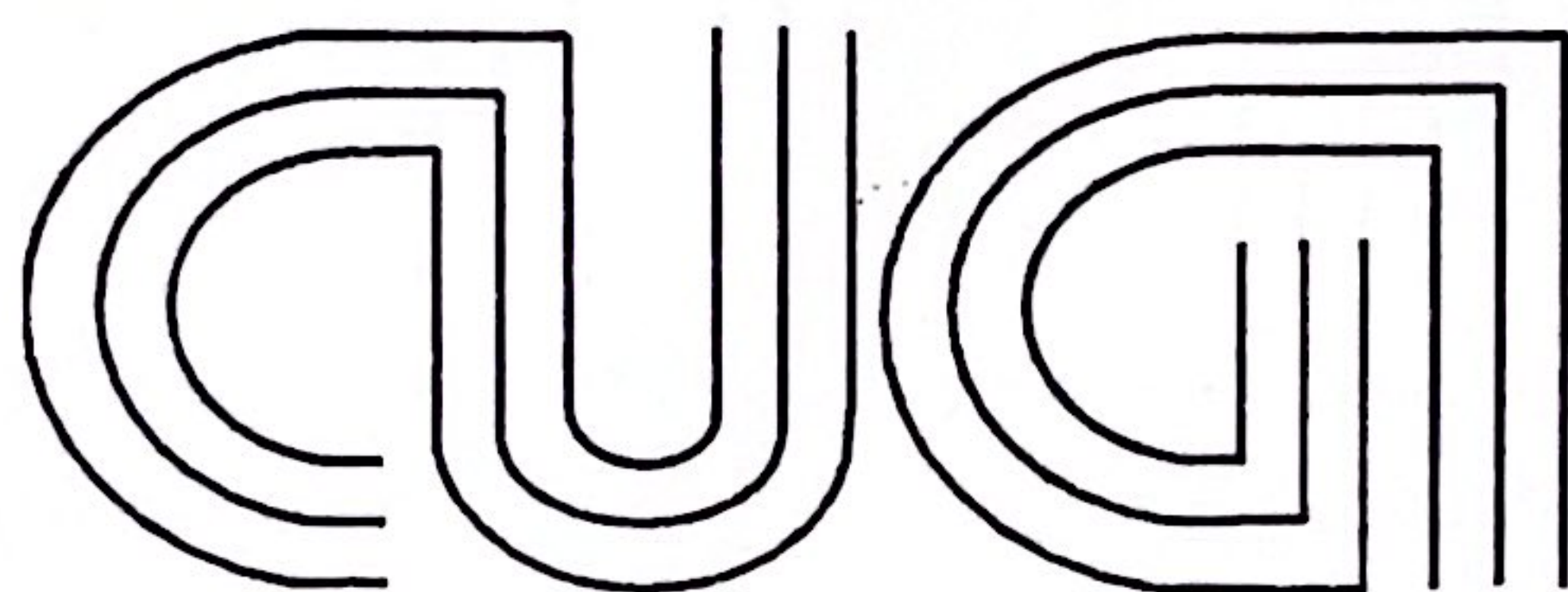




Newsletter

September 1990

In this issue...

Setting up the A590

Irish Bulletin Boards

Objected Oriented Programming

Workbench Disk Revealed

Natural Language Parsing

HONORARY CUGI OFFICIALS

Chairman	Geoffrey J. Reeves
Treasurer	Brian Ward
Membership Secretary	Shane Broadberry
Chief Librarian	Thomas Kinsella
Amiga Librarian	Rocco Matassa
PC Librarian	Brian Ward
C64 Librarian	Stephen Kemp
C128 Librarian	Geoffrey J. Reeves
Communications	Liam Murphy
Newsletter Editor	Edward Carroll

CUGI meetings are held every second Friday, at 8:00 PM in the computer room at St. Andrew's College Booterstown. All correspondence should be addressed to:

CUGI
c/o St. Andrew's College,
Booterstown,
Blackrock,
Co. Dublin.

All articles in this newsletter are copyrighted by the author unless otherwise stated. If you wish to reprint an article, write to its author c/o CUGI at the above address.



Newsletter

Contents

Volume 2 Number 4
September 1990

Editorial	Eddy Carroll	2
Natural Language Parsing	Geoffrey J. Reeves	5
Bulletin Boards	Brian Ward	8
C Sick -	Eddy Carroll	13
Setting up the A590	Leon Hurst	18
Amiga 3000 Hints and Tips	Eddy Carroll	22
The Message	Colm O'Rourke	24
Object Oriented Programming	Russell Wallace	26
Making the Right Connection	Geoffrey J. Reeves	38
SCART Leads	Tom Kinsella	39
A Tour of a Workbench Disk	Shane Broadberry	42

Editorial

It seems like ages since I last produced a full issue of the newsletter! Geoff kindly took over as editor for the March issue, when I was busy studying for my finals, and he then got landed with the task again for the June issue when I unexpectedly had to go off to America for 5 weeks. I think you'll agree he coped admirably.

Yet again, this issue is our biggest to date. Russell has surpassed even his own standards with a detailed article on object-oriented programming, currently the in thing in programming circles. Brian looks at bulletin boards, Colm provides us with another entertaining short story, Geoff looks at ways of getting your micro to understand English, Shane takes a tour around a typical Workbench disk, Tom explains how to put together a SCART lead, and Leon gives some essential tips for getting your new A590 up and running as painlessly as possible. For your entertainment (the C programmers among you anyway) I've uncovered some C code you wouldn't like to meet alone on a dark night. All in all, there should be something for everyone. Yet again, no C64-specific articles; I'm assuming that the lack of complaints about this means that they aren't being missed.

A fair amount has happened since my December editorial. Probably the most significant change (as Geoff hinted at in the last issue) is that I have now left college and have to do real work for a change. The company innocent enough to employ me is Artist Graphics Europe, which is based in BIM House opposite the East Pier in Dun Laoghaire. It's actually the European subsidiary of the American company Artist Graphics which in turn falls under the umbrella of the Control Systems group of companies. Their business is making high resolution graphics boards for the PC CAD market (by high resolution, I mean starting from 1024 x 768 up to 1600 x 1200, with a palette of 16 million colours. The Amiga has some catching up to do I'm afraid). In order for these boards to be of any use, they need software that allows popular PC CAD software to be used with them, and this is where I come in. My current project is writing a driver for a high end 3D architectural package called ARC+.

Working with genuine Intel-based PC's has been quite revealing; I still wouldn't swap my Amiga for a PC for any amount of money, but they have become a lot more powerful than I had realised. Now that I've actually had to program the 8086 chip, I can safely say that all those people who criticise it are completely justified. As a well known Usenet poster says, it

is impossible to both understand and appreciate Intel CPUs. On the other hand, I've been learning a huge amount over the past few months which has been a lot of fun. It's not all Intel hardware either; the particular graphics board I'm working with has an onboard Texas Instruments 34020 processor. This can be thought of as a 68020 and a blitter rolled into one, with a lot of other nice bits added in. Development tools exist on the PC to write code that runs directly on the 34020. The graphics board has 2 megs of video ram and 5 megs of dynamic ram and also executes code faster than the 80386, so in many ways, the PC is just acting as a dumb terminal interfaced to the graphics board. I find this quite amusing.

One of the nicer perks about joining Artist Graphics was that they have a policy of sending all new employees over to their headquarters in St. Paul, Minnesota for training. Hence the sudden disappearance to the US I mentioned above. I was there for the month of July and the first week of August, and I had a really good time. Apart from the fact that all expenses were paid (of course), I arrived in time for the annual company holiday which takes place at a holiday resort called Maddens in North Minnesota. The whole company plus spouses (about 200 people altogether) headed up there on a Friday afternoon, until the following Sunday evening. It was fabulous. I also got an opportunity to go white water rafting at Lake Superior, which I would highly recommend to anyone who finds themselves in the area! I'll refrain from making you all jealous by relating the prices of consumer goods in the states, other than saying that in most cases, the American price of an item is calculated by taking its Irish price and replacing the pound sign with a dollar sign.

But enough about how tough my first few months of work have been (grin). What else has been happening? Well, I got my hands on an early Amiga 3000 in the middle of June, as you know from the last newsletter. The new job hasn't left an awful lot of time for playing with it, but I've detailed the things I've discovered so far in an article elsewhere. My review last time was a little rushed I'm afraid but I'm sure you got the idea that I was fairly impressed.

One of the things my A3000 is doing is acting as a UUCP site. UUCP is the underlying protocol used to send Usenet news around the world, and Commodore Europe have adopted the standard as the basis of their private developer network. Once or twice a week, my A3000 rings up Commodore UK's Unix system and downloads the latest messages and files from European Amiga developers. It all happens unattended, in the middle of the night when phone rates are cheapest.

While this doesn't link me into the main Usenet network, there may soon be an easy way to access Usenet cheaply from Dublin. For those of you not quite sure what Usenet is all about, it's essentially a network of tens of thousands of Unix systems around the world. Distributed among these systems is a global message database, divided into about 800 newsgroups covering all manner of things from Amigas to cars to compact discs to the environment. A message posted to a particular newsgroup will typically be seen everywhere in the world within a few days. Around 4 megabytes of messages per day are generated, and while there is a fair amount of 'noise' (worthless material) there is also a good amount of valuable information. In addition, there are newsgroups which carry only program files. These are ideal for getting the newest public domain software for your computer.

Declan McArdle, together with a group of Dublin BBS sysops, is in the process of organising a public access Usenet site, which would allow anyone with a modem to read and post Usenet news and international email. The cost is expected to be in the region of £50 per year. A public meeting is planned for 6th October at a venue yet to be decided, to discuss things further; all interested are welcome. Contact Declan on 884195 for further details.

Many of you have probably heard of Jeff Walker. Not only was he an editor at Amiga Computing, he also took the CUGI committee out to lunch at last year's Commodore Show! He has now left Amiga Computing and started his club magazine, Just Amiga Monthly (JAM). It covers all Amiga topics except games (At last! A non-games mag!). A year's subscription is Stg.£29.95, but you can get it for just £19.95 if you mention the fact that you're a caller to Infomatique BBS. The first issue arrived through my door a few weeks ago and it seems to have got off to a fairly good start. 40 relatively ad-free A4 pages include articles on the KCS PC emulator for the A500, Commodore's CDTV, and how to use double buffering from assembly language, just to pick a few at random. The address to write to is: Just Amiga Monthly, 2 Sandbourne Road, Birmingham B8 3NT.

Out of time and out of space, as usual. Don't forget the AGM which takes place this year on October 26. Remember that you can nominate someone to cast a proxy vote on your behalf if you won't be able to make it on the night. The only other thing remaining is to remind you all that the deadline for articles for the next newsletter is 23 November. Until next time, enjoy this issue.

E.C.

Natural Language Parsing

by Geoffrey J. Reeves

Anyone who has been taught (is that the same as learned?) a second language will know how difficult it can be to make sense of a sentence. One usually tries to split it up into words, to translate each word and hope that a perfect reconstruction in English will follow. Needless to say (but I'll say it anyway), that rarely works. It fails because individual words often do not translate well out of context. At this stage, it is worth looking at our own language to see just how complicated it is.

I suspect we have all been through parsing of English sentences – working out the various clauses, nouns, predicates, prepositions, etc. What I am presenting in this article is a system that evolved over a number of years which makes an attempt at simple parsing of simple sentences. Its purpose? Adventure games for one, an exercise in *Natural Language Recognition* for another. The latter is a part of the subject *Artificial Intelligence*.

Before I begin, I must make it clear that this article is not necessarily aimed at programmers. Such people are, of course, welcome to try out the ideas presented. Others should find enough food for thought (translate that one into a foreign language!) in the ideas alone. Okay? Right – let's start with the language itself and a simple sentence:

TAKE THE IRON BAR AND WRAP IT AROUND FRED'S NECK

Simple? No chance! You know exactly what it means because you can read the complete sentence. You can easily categorise the various words into nouns, adjectives, verbs, prepositions, articles, conjunctions and so on. But do you realise how many of the words fell into several categories?

Iron - a noun (as in an ironing board)
- an adjective (an iron girder)
- a verb (back to the ironing board again)

Bar - a noun (where you drink, or like a rod)
- a verb (as in "You can't!")

So, the first thing I did was to set up a vocabulary list containing all the words I wished to recognise. Ideally, this should have been the full list in alphabetical order, but as the project which required this parsing was evolving continuously, this was not possible. Besides which, it did seem that the idea of breaking the vocabulary up into segments (of a previously determined size) was a good idea. For example, one could set up the following:

Word Type	Start	End	Total
Adjectives	1	100	100
Prepositions	101	120	20
Conjunctions	121	140	20
Articles	141	150	10
Pronouns	151	160	10
Other words	161		Unlimited

You will notice that I did not distinguish between nouns and verbs because I found that there were too many words that fitted into both categories. The other reason was that the project was intended as a parser for a text adventure and as such was an extended two-word command parser. By this, I mean that the adventure would primarily respond to two-word commands such as *go north*, *eat biscuit*, *kill dragon*, *kiss princess* and, obviously, one-word commands such as *look*, *inventory*, *help*, *sigh*. I did not want to have a problem with words such as *light* and *water* which might be used as *light torch* and *extinguish light* or *water plant* and *drink water*.

I set myself the goal of parsing (and properly dealing with) a user instruction of the form:

PUT THE RUSTY KNIFE INTO THE LARGE SACK AND GO EAST

My basic algorithm for parsing this would be:

1. Is there a conjunction? This would mean at least two instructions.
2. Is there a preposition? No preposition would mean a fairly straightforward parse.
3. The instruction or that part before the preposition should only contain two important words, or three if there is an adjective.
4. The part after the preposition should contain only one word, or two if there's a second adjective.

So what did I mean by 'important words'? Simply, words which were not 'the' or 'a' or 'an'. This reduces the first part of the above example to:

PUT RUSTY KNIFE INTO LARGE SACK

The parsing would then perform a check (look up word in vocabulary and find to which category it belonged) to produce this list:

word adjective word preposition adjective word

At this stage, various checks must be made to check the validity of the sentence. For example, a sentence may not end with a preposition nor may it begin with one. Mind you, I am aware that at some stage a player may enter *Hit Fred*, bringing the response *With what?* whereupon the player replies *With the brick*. Hmmm...

If the sentence passes the validity checks, my plan is to convert the two words into a single number. Since each word has a unique position in the vocabulary, this could be done unambiguously by multiplying the first word's position by (say) 10000 and then adding the position of the second word. I can then rapidly check my list of acceptable two-word instructions (I call these *events*) and process the event accordingly. The adjective and prepositional part can be checked for (somewhat awkwardly at the moment) and the complete instruction executed.

The checking of adjectives provided a few interesting problems. For example, some objects in an adventure do not have a specific adjective describing them. If the user gives an adjective, it should therefore be ignored, even if it is rather silly (*drink fat water*, for example). If no adjective is entered and many similar objects exist, then the response *Which one?* is needed. I'm sure this mucks up my idea of two-word events.

In the meantime, I'd better get back to the practical side of this – the actual programming! By the way, if you are interested in reading about what I've only touched upon, try these:

Adventure Description Language

A.F. de Geus, J.H. Jongejan, A.M. Koelmans,
Sigma Press.

Artificial Intelligence and Computer Games

Richard Bartle (author of the original MUD),
Century Communications.

For Sale

Panasonic KX-P1180 9-pin dot matrix printer. Never used. Includes several internal fonts, paper parking, and LCD front panel display, with facilities for changing settings directly using control buttons. £250 o.n.o. Contact Eddy Carroll on 874540.

Bulletin Boards

by Brian Ward

I know there are many of you out there who already know what a bulletin board looks like and what to do when you call one so I must point out now that this article is not aimed at you. It is aimed at those people who have recently bought a modem, or are thinking of buying one in the near future.

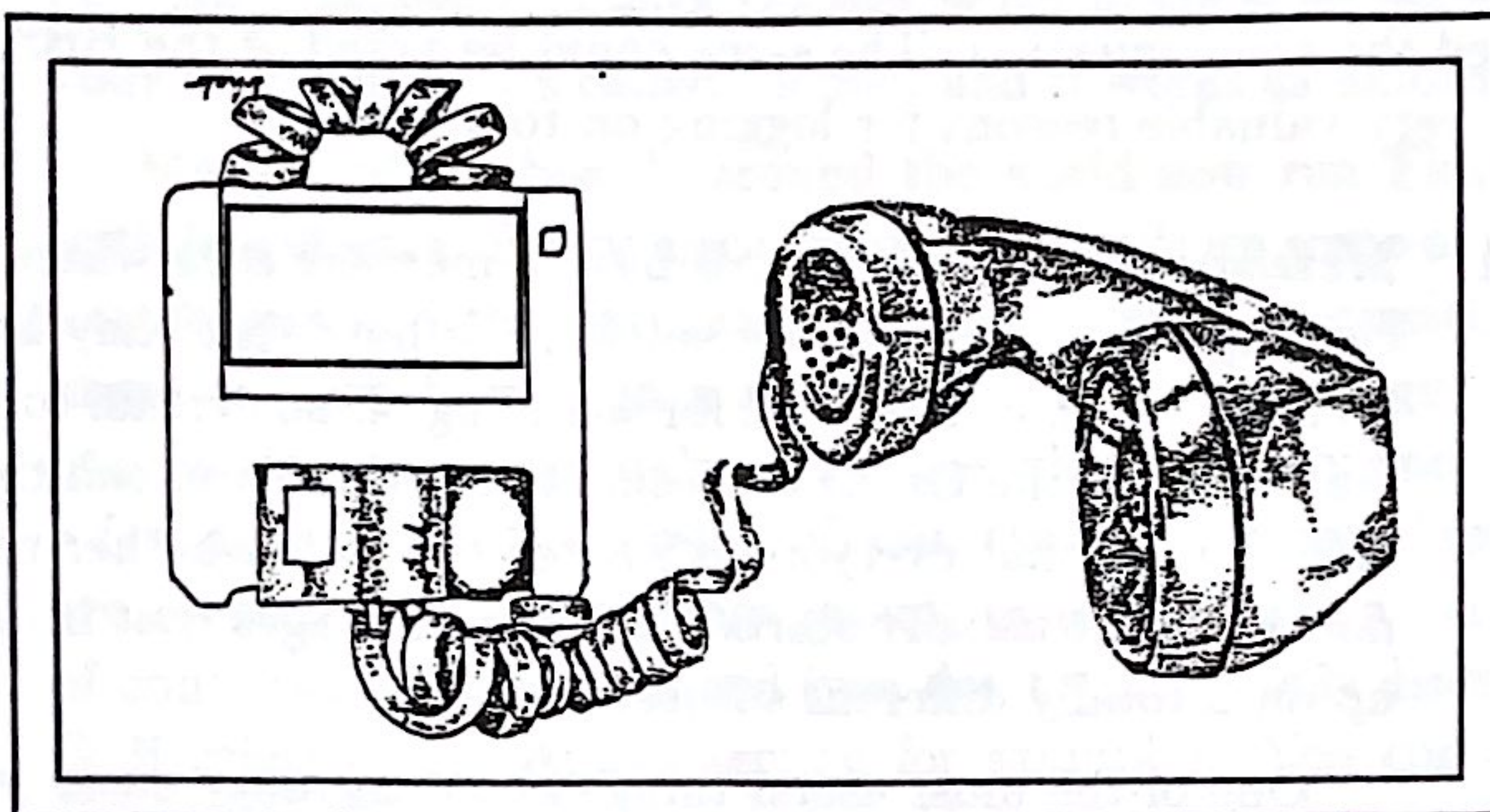
Modems

There are as many modems as there are computers; once again, the dilemma arises of which one to buy and what speeds to get. The first thing to decipher is what is a baud. This is the speed at which data is transferred between computers, so the higher the baud rate, the faster that you can upload files or programs. Unfortunately, the price can be directly related to the speed, but if you hunt around, you'll find one that will be fast enough without breaking the bank. Ideally, a 9600 baud intelligent modem would be your choice, but if funds won't allow this then you can get by quite adequately with a 2400, 1200 or even 1200/75 baud modem.

You'll also come across the V numbers. These refer to various standards in the modem world. The most common are V21 (300 baud), V22 (1200 baud), V22bis (2400 baud), V23 (1200 baud receive, 75 baud send), V32 (9600 baud), and V24 (confusingly, the name of the RS232 connector that you find on your modem and computer). Note that V32 is just one of the 9600 baud standards currently around; it will likely prevail in the long term but it is not all that common yet.

Some moderately priced modems are the Courier HST 9600, the Transmodem 2400 and the Hitec EC2400 Plus. The Hitec supports MNP/5 which essentially means that when you're calling a compatible modem (most bulletin boards in Dublin support MNP/5) you can get a transmission rate of up to 4800 baud over a 2400 baud link (don't ask how, just take my word for it). All these modems run with virtually any computer with the right cable. If you have a C64 the Compunet modem is excellent value, or you could try the MultiModem.

Nowadays, you can get a modem that can also double as a fax machine. A word of warning though: do NOT buy your modem over here, unless you can really afford it (i.e. your company is paying). Some Irish companies seem to rip off any and all unsuspecting buyers because they don't know what they're looking for or how much it should cost. My advice is to import from the UK and you could make a saving of up to 50%.



Software

As with the modem, there is a plethora of communications software available. For the Amiga, I would recommend *JR-Comm*, which is a shareware program and costs next to nothing. For the PC, I would recommend *PCPlus* or *Procomm*, also shareware.

History

The first bulletin board is reputed to have started in the late 70's in California (where else). It ran at 300 baud and I believe it is still running today at 300 with lots of callers. Roughly around the same time, a student in Essex University, England, decided to write a multi-user dungeon as his 4th year project. Other students asked if they could play the game and it soon gained a strong following. In 1984, it was decided to make the game public and so MUD became available to the masses.

From then on people like British Telecom, Compunet and CompuServe discovered that this was a very easy way to make lots of money, and so they set up their own versions of MUD. Naturally, the public weren't thrilled about having to pay for the privilege of MUDding, and quite a few private MUDs were setup on bulletin board systems around England. The more well known ones were Wanderland, Gods and Shades (Shades later migrated to Micronet, where it became very popular indeed, but gained a charge of several pence/minute). Unfortunately, there are no realtime multi-user games available via Irish bulletin boards ...yet.

Uses for Bulletin Boards

One of the first questions I got asked when I bought my first computer was, "What can it do?" I'm sure most of you reading this article must have been asked the same question. The same could be asked of the BBS. Here are a few very valuable reasons for logging on to a board.

1. *Messages.* Almost all boards have a message area where individuals can express opinions about software or hardware they have recently bought. It can also be used for sounding off on certain topics, like the Gulf crisis or the Ozone depletion or anything you can think of. The nice thing is that everyone is invited to reply, whether they agree or not, and that usually starts a train of messages that invariably ends up on a totally different subject.

One of the most useful things about message bases is that when you have a problem with hardware or software, or if you are looking for some advice on purchasing same, you just upload your query into the appropriate section, and it is almost guaranteed that you'll have at least one answer by the next day, all for the price of a local phone call. Every board has its own resident buffs who are expert in a particular field or on a certain machine, and there is little you can ask that will not be answered by someone.

Most boards also provide a facility for private messages, called electronic mail (EMail). You can leave several confidential messages to your friends in a single call, and they will read them the next time they are on. No problems with nobody being home when you call, and much faster and cheaper than the post.

2. *Software.* Probably the best place in any country to find public domain or freeware programs is on a bulletin board. A few boards insist on a one for one basis (i.e. you must upload one program for each program you download) but this is rare. Most boards operate on the basis that once you've been cleared as a bona fide user, you can download to your heart's content. In a lot of cases, PD programs can rival their commercial counterparts. Also, in a lot of cases these days, a private programmer will write a program as a college project to emulate a better known package, and can end up with a better version of the same package. Since he/she doesn't have the same advertising power as the Mega Corporation, the program is released as freeware with a request for a subscription towards the cost of providing a decent manual with updates; this is usually about \$25.

3. *Contacts.* The best place of all to find out about clubs or other user groups is on a bulletin board. There is now also a means whereby you can communicate with the rest of the world from the comfort of your own home. It's called Fidonet and it works as follows.

Many bulletin boards around the world now run Fidonet compatible software. Every night, each board calls up some of its closest neighbours and they exchange messages posted to special *echomail* areas on each board. In a short space of time, messages posted to one board can spread right across the whole country and after only a few days will have travelled all over the world. Special moderators are elected for each message group, to ensure that a certain level of conversation is maintained (you don't want people discussing the Gulf crisis in the Amiga section for example). You can also send private mail across the network, which can work out much cheaper than making a long distance phone call.

Usenet, the network connecting Unix systems around the world, operates in a very similar manner, but at a somewhat higher level of technology. Many Usenet sites have permanent leased lines to their neighbours, and the hardware they run on tends to be more expensive as well.

The Future

Well, if you thought the modem was a thing for other people, you may find yourself having to think again. In October, the official launch of Minitel is due. Minitel is a French idea which is now being brought over to Ireland. In France a few years ago, the phone company, France Telecom, decided to try out an idea. Instead of printing telephone directories, they issued to each subscriber, free of charge, a small computer terminal with a built in modem. The idea was that they wouldn't have the expense of printing several million phone books each year, which were usually out of date by the time they were received, and the customer could have access to the constantly updated computer files of phone numbers.

The idea not only worked, it exploded. Within a short space of time, there were many online services available in addition to the standard electronic directory. So successful has the system been that you can now check your bank balance and move funds, reserve a flight, or book a theatre ticket. You can call up a map of France on the screen, tell the computer where you are and where you want to go, and then ask it for the suggested roads, or if you want to avoid the toll roads, a suggested journey using non-toll

roads and what your ETA would be. These are only a few of the services available to the French.

Now a consortium of about 20 companies have got together, and the first terminals should be in the country already. Distribution is expected to start around mid October. Initially, there are about 6000 terminals, which are being supplied by Phillips, and depending on reaction, more will follow. Among the 20 companies are AIB, Telecom Eireann and Family Album. It is hoped that services such as teleshopping, tourist information, agricultural advisory services and travel timetables will be ready for the launch. The company running this system is called Minitel Communications, and they hope to have about 30 services online by the end of the year, and over 300 by the turn of the century. The terminals will cost about £5 per month and will be available from Telecom Eireann.

Apart from this, telecommunications is becoming easier and cheaper all the time. If you need information fast, the best way to get it is via the phone. If you have a database of information, you can cut down wages and phone bills by buying a modem and giving people access to the information. There is now an online service from the biggest credit information company in the world, Dun & Bradstreet, where you can dial up and get information about a company such as their last year's accounts, director's names, turnover etc., all within minutes. Admittedly, this costs about £30-£50 a report, depending on the amount of information that you require, but if you have to make a decision on whether to give credit on a multi thousand pound contract, money like that is well spent. A word of warning though. There are credit reporting agencies that thrive on peoples ignorance. I was quoted £350 consultancy fee for advice on what modem and software to buy in order to go online. This did not include the modem or software, just advice!

So, if you do own a modem or are contemplating buying one, here is a list of the public bulletin boards available in Ireland. If you know of any I left out, please contact me through the club so we can add them to the list.

Name	Phone No.	Speeds	Time
Infomatique	302970	300-9600	24 hours
Niteline	834660	300-2400	24 hours
TOPPSI	711047	300-2400	24 hours
Dubbs, line 1	885634	300-2400	24 hours
Dubbs, line 2	831908	2400-9600	24 hours
ICGPBBS	785866	300-2400	6pm-9am

Local Dublin Bulletin Boards

C Sick

by Eddy Carroll

Last time, Shane explained how to structure your C programs to make them easier to debug and maintain. For a change, I thought I'd head off in the opposite direction and look at the sort of warped C code that can result when you really put your mind to it. Non programmers can tag along for the ride as well; you might find it fun.

Before I start though, a little puzzle: write a C program that prints out a copy of itself when you run it. Copying the source file to the screen is not allowed, of course. Answer at the end of this article.

Every year on Usenet, the International Obfuscated C Code Contest takes place. Its goals are to show what should *not* be done in C programs, and to provide a forum for poor C code. The only restrictions on entries are that they should be as obfuscated as possible, and no larger than 1536 bytes in size. As an example of the sort of insanity this provokes, look at the following code by Brian Westley of St. Paul, Minnesota; it won the 1989 award for most algorithms in one program.

```
/**/};);/**/main(/**/tang ,gnat/**/ABBA",0-0(avnz;)0-0,tang,raeN
,ABBA(niam&&))2-]--tang-[kri - =raeN(&&0<)/*clerk*/,noon,raeN){(!tang&&
noon!=1&&(gnat&&2)&&((raeN&&( getchar(noon+0)))||1-raeN&&(trgpune(noon
))))||tang&&zava(/**/tang ,tang,tang/**/((||))0(enupgrt=raeN
(&&tang!(||))0(rahcteg=raeN( &&1==tang((&&1~)gnat=raeN(;;)tang,gnat
,ABBA,0(avnz;)gnat:46+]552&)191+gnat([kri?0>]652%)191+gnat([kri=gnat
(&&)1~gnat(&&)1& ABBA(!;)raeN,tang,gnat,ABBA(avnz&&0>ABBA{)raeN
,/**/);}zava(/**/tang,gnat,ABBA/**/7(niam;)-1-,78-,611-,321
-,321-,001-,64-,43-,801-,001-,301-,321-,511-,53-,54,44,34,24
,14,04,93,83,73,63,53,43,33,85,75,65,55,45,35,25,15,05,94,84
,74,64,0,0,0,0,0,0,/**/){ABBA='N'==65;(ABBA&&(gnat=trgpune
(0)))||(!ABBA&&(gnat=getchar(0-0)));(--tang&&1)&&(gnat='n'<=
gnat&&gnat<='z' || 'a'<=gnat&&gnat<='m' || 'N'<=gnat&&gnat<='Z'
|| 'A'<=gnat&&gnat<='M'?((gnat&&/**/31/**/21,11,01,9,8
,7,6,5,4,3,2,1,62,52,42,/**/)+12)%26)+(gnat&&/**/32/**/22,12,02,91,81,71,61,51,41{=]652[kri];)*pry*/)+65:gnat);main
(/**/tang~tang/**/ ,/* ,~/**/tang,gnat,ABBA-
0/**/((niam&&ABBA||)))tang( rahcteg&&1-1=enrA(||))tang(
enupgrt&&1==enrA((&&)2&gnat(&& )1~tang(&&ABBA!(;)85- =tang
(&&)'a'<=gnat(&&)1==gnat(&&)4 ==ABBA(&&tang!;)))0(enupgrt=
gnat(&&tang((||))0(rahcteg =gnat(&&tang((&&ABBA;)-1-'A'!=
'Z'=tang(&&ABBA{)enrA/**/);gnat ~-1&&zava(tang+1,gnat,1+gnat);
main(ABBA&&2/**/\\**/,tang,gnat ,ABBA/**/7(avnz/**/);}/**/
```

Now, I'm sure you're all saying to yourselves, "Of course! It's a filter program!" and indeed, that is exactly what it is. In fact, it's a program that

will read a file from standard input, contort it somewhat, and output the result to standard output. The way the input is altered is controlled by the number of parameters given to the program when you run it.

With no parameters, the program simply copies input to output unchanged. With one parameter, it ROT13's the input.¹ With two parameters, it reverses the input (i.e. outputs the last character first and the first character last). Finally, giving three parameters both ROT13's and reverses the input.

So far, nothing to justify the above mess, you might be thinking. The clever bit is that if you feed the source code itself into the program, it outputs a different program that does exactly the same thing! Not only that, but whereas the original source look like a chunky letter K, two of the transformed versions look like the letter R. These are of course the initials of Brian Kernighan and Dennis Ritchie, who designed the C language.

In fact, Brian Westley turns up again in the 1990 contest, this time with the prize for Best Program Layout:

```
char*lie;
    double time, me= !OXFACE,
    not; int rested,  get, out;
    main(ly, die) char ly, **die ;{
        signed char lotte,
```

```
dear; (char)lotte--;
    for(get= !me;; not){
        1 - out & out ;lie;{
        char lotte, my= dear,
        **let= !!me *!not+ ++die;
        (char*)(lie=
```

```
"The gloves are OFF this time, I detest you, snot\n\Osed GEEK!");
do {not= *lie++ & 0xF00L* !me;
#define love (char*)lie -
love is *(not= atoi(let
[get -me?
(char)lotte-
```

```
(char)lotte: my- *love -
'I' - *love - 'U' -
'I' - (long) - 4 - 'U' ])- !!
(time =out= 'a')));} while( my - dear
&& 'I'-11 -get- 'a'); break;}}
(char)*lie++;
```

¹ROT13 is a simple encryption mechanism used on Usenet which replaces A with N, B with O and so on. Non-alphabetic characters are left unchanged.


```

(char)*lie++, (char)*lie++; hell:0, (char)*lie;
    get *out* (shortly -0-'R'- get- 'a'-rested;
    do {auto*eroticism,
    that; puts(*( out
        - 'c'
    -('P'-'S') +die+ -2 ));}while(!"you're at it");

for (*(char*)&lotte)~=
    (char)lotte; (love ly) [(char)++lotte+
    !!0xBABE];){ if ('I' -lie[ 2 +(char)lotte]){ 'I'-11 ***die; }
    else{ if ('I' * get *out* ('I'-11 ***die[ 2 ])) (*(char*)&lotte) -=
    '4' - ('I'-11); not; for(get=!)

get; !out; (char)*lie & 0xD0- !not) return!!
    (char)lotte;}

(char)lotte;
    do{ not* putchar(lie [out
    *!not* !!me +(char)lotte]);
    not; for(;!'a';);}while(
        love (char*)lie);{

register this; switch( (char)lie
    [(char)lotte] -1s *!out) {
    char*les, get= 0xFF, my; case' ':
    (*(char*)&lotte) += 15; !not +(char)*lie*'s';
    this +1s+ not; default: 0xF +(char*)lie;}}
    get - !out;
    if (not--)
    goto hell;
    exit( (char)lotte);}

```

Now, who says C isn't readable? Oh, you'd like to know what the program DOES? Well, it actually simulates plucking the petals from a daisy. More precisely, you run it with a single parameter which is the number of petals to pluck, and it prints out lots of "westley loves me" and "westley loves me, not" messages. See if you can find the "westley" string hidden in the code.

Another nicely formatted program came from Byron Rakitzis and Sean Dorward, who won the Best Utility award for 1990. Note the cute sine wave along the top, not to mention the familiar number down the left hand side:


```

#define D ,close(

char          *c,q          [512          ],m[          256
],*v[          99], **u,          *i[3];int          f[2],p;main          (){for
(m[m          [60]=          m[62          ]=32          ]=m[*          m=124          [m]=          9]=6;
e(-8)          ,gets          (1+(          c=q          )||          exit          (0);          r(0,0)
)for(          ;+++          c); }r(t,          o){          *i=i          [2]=          0;for
(u=v          +98          ;m[*--c]          ^9;m          [*c]          &32          ?i[*c
&2]=          *u,u-          v^98          &&++u:

3          )if(!m[*c]){for(++c=0;!m[*--c];);
*          --u= ++c;}u-v^98?strcmp(*u,"cd"?*c?pipe(f),o=f[
1          ]:
4          ,(p=fork())?e(p),o?r(o,0)D o)D*f):
1          ,wait(0):(o?dup2(*f,0)D*f)D o):*i?
5          D 0),e(open(*i,0)):
9          ,t?dup2(t,1)D t):i[
2          ]?
6          D 1),e(creat(i[2],438)):
5          ,e(execvp(*u,u)):e(chdir(u[1])*2):
3          ;}e(x){x<0?write(2,"?\n$ "-x/4,2),x+1||exit(1):
5          ;}

```

For those of you who haven't guessed, the above program implements a minimal Unix shell, which supports I/O redirection, pipes and current directories.

There are many other winners equally worthy of mention, but space precludes them all being listed. You can however find archives containing the complete collection on Infomatique, in the files area. Alternatively, ask me to bring them along to a CUGI meeting some evening.

Of course, a program doesn't have to be obfuscated to be different. The following program is written in street-wise C, source unknown. Remember kids, don't try this at home without adult supervision!

```

/*
 * Just wanted to check out that you gnarly dudes are using the latest
 * and greatest software technology fer yer rad code to make it easy for
 * the dudes who have to read it. The hip new way to write readable C code
 * involves the use of a few simple defines.
 */

```

```

#define like {
#define man ;}
#define an ;
#define SayBro /*
#define CheckItOut */

```

SayBro like, this is some rad shit, so CheckItOut


```

like
    a = b
        an
    c = d
man

```

SayBro , like who needs help from them compiler choads anyway?
 THIS is the way to write CLEAR code. I mean really! CheckItOut

like SayBro this is ShellSort straight out of the white book, but in
 a readable form.

CheckItOut man

```

#define YoDude for(
#define OK      )
#define is      =
#define AND     &&
#define as
#define Do
#define long
#define some
#define make
#define shit
#define FAROUT

shell(v, n) SayBro sort v[0]...v[n-1] into increasing order CheckItOut
int v[], n;

```

like int gap, i, j, temp;

```

YoDude gap is n/2 an as long as gap > 0 Do some shit an make gap /=2 OK
YoDude i is gap an as long as i < n Do some shit an make i++ OK
YoDude j is i - gap an as long as j >= 0 AND v[j] > v[j+gap] Do some
shit an make j -= gap OK
like
    temp is v[j]      an
    v[j] is v[j+gap]  an
    v[j+gap] is temp
man
FAROUT man

```

SayBro like, B there 0B square! CheckItOut

So, now you have an idea of just what fun you can have with C and its
 preprocessor. I hope you found the above programs amusing. Believe me,
 you have to get very good at writing standard C before you can start to
 abuse the rules with such disregard. To finish off, here's the answer to that
 puzzle I set at the start: A C program that prints itself out. Enjoy!

```

char*p="char*p=%c%s%c;main(){printf(p,34,p,34);}";main(){printf(p,34,p,34);}

```


Setting up the A590

by Leon Hurst

It is only now that I realise that had I followed a set format when setting up my new A590, it would only have taken me two hours and not two weeks (honestly). I am now going to give you a simple set of instructions by which to set up a hard drive. However, not all of the format will be needed, depending on your needs.

If you want to partition the drive or create any virtual floppies then have a look at the User's Guide and December '89 CUGI Newsletter before proceeding. However, if you have not altered the hard drive, all the files found in Workbench 1.3 will already be installed and so you can skip the next paragraph.

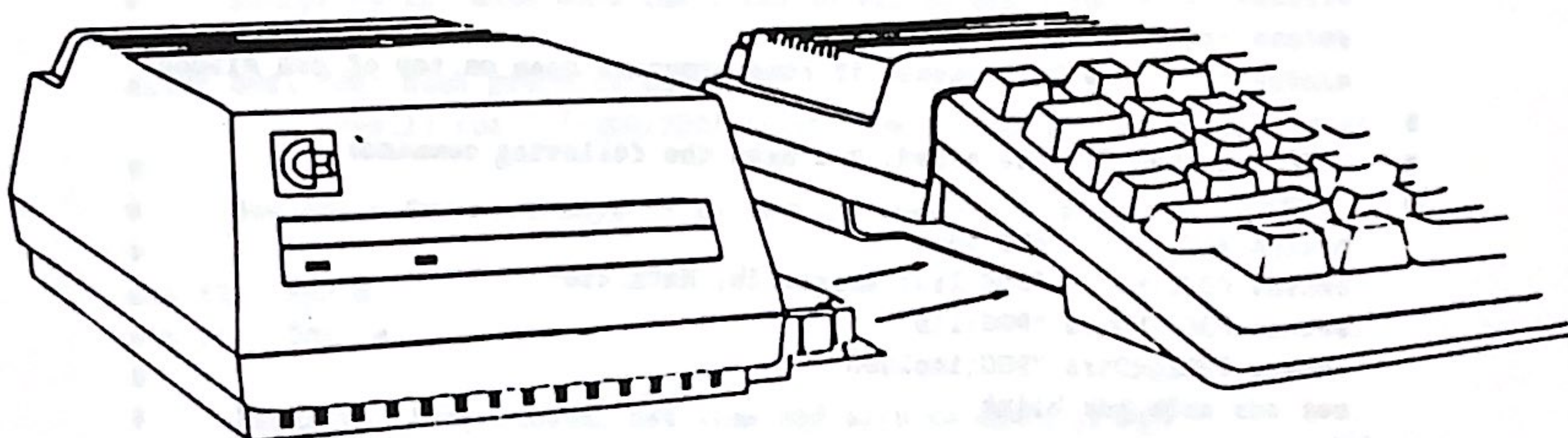
The idea now is to get all the files found on the Workbench 1.3 disk onto the hard drive. Either using a CLI or Workbench, copy the *InstallHD* program from the *A590 Setup* disk to the relevant partition, CD to that partition, run the program and follow its instructions (unless you don't want *AmigaBasic* in which case you click CANCEL when a requester for the *Extras 1.3* disk appears). You now have a hard drive which you can boot from. Reset the computer.

For reasons of maintenance, one cannot successfully operate a hard drive in a WIMP environment such as Workbench (at least, not until Workbench 2.0 becomes available). So, you must enter the world of CLI. The CLI in the System drawer is very user unfriendly and an enhancer such as *CShell V4.01A* (on CUGI Disk Programs #1) is essential. Just copy the *CShell* directory and its contents from that disk onto the hard drive. The *arp.library* from the *LIBS* directory should also be on the hard drive - if not, copy it across. Using a text editor or a word processor, edit the startup-sequence by replacing *LoadWB* with *Cshell/Shell*. Then reset the computer.

MAKE A BACKUP OF THE HARD DRIVE NOW AS INSURANCE AGAINST ACTS OF GOD AND THE DELETE * COMMAND.

The hard drive is now fully operational. But to more easily gain the extra facilities which *CShell* offers, we must create a file called *Startup.sh*. It really replaces what normally goes in your startup-sequence, which need now contain only the following:

```
Setpatch r ; For one Meg chip ram + RAD users
CShell/shell s/startup.sh
```

Commodore's A590: More power for the serious user.

The idea of this is to execute a startup.sh file which contains what was in the original startup-sequence and much more. The following is a list, with comments, of the entries to the startup.sh file. Anything from a '#' to the end of the line is treated as a comment.

```

window -l -a                # Make window full screen size
set _prompt  ^[[33m%p>^[[0m;    # A fancy prompt in colour
set _titlebar "CSH V4.01A -- Amiga CLI/Shell replacement"
cd dh0:
#
#   The following lines are executed only when you first boot up your
#   Amiga; if you subsequently re-execute this script, they will be
#   skipped. This is because they only need to be run once.
#
if -n $startup = 1
    set startup 1
    Path c: dh0:bin dh0:system dh0:pdcc dh0:cshell401    # Standard path
    FastMemFirst      # If any fast memory in A590, make it default
    setmap gb          # Make sure we use the english keyboard layout
    stack 20000        # Lots of programs work better with a big stack
    mkdir ram:env       # These three directories are used by WB 1.3
    mkdir ram:t         #
    mkdir ram:clips     #
    assign env: ram:env  # And these three assigns are needed too
    assign t: ram:t     #

```



```

assign clips: ram:clips #
mount ff0:                # You need this only if you have a virtual floppy
#
#   Add any resident utilities etc. you want in here.
#
virusx4 -c -r
dmouse -a8
window -f                # Only needed if some programs open on top of csh window
#
#   If you have PDC installed, you need the following commands
#
assign PDC:                dh0:pdcc
setenv PDCLibs             "PDC.lib; Amiga.lib; Math.lib"
setenv PDCLibDirs          "PDC:lib"
setenv PDCIncDirs          "PDC:include"
rez ccx a68k pdcc blink
endif;
#
#   Here ends all the once-off commands. Now follows a list of popular
#   abbreviations for often used commands.
#
alias Bye                  "exit; exit; exit" # Quits CShell (must type capital B)
alias copyall              "copy -r "        # Copy everything in curdir and subdirs
alias ctp                  "ChangeTaskPri"   # Change priority of shell
alias dq                   "dmouse quit"     # Quits DMouse if you have it running
alias ds                   "dmouse -a2"      # Slows down the pointer using DMouse
alias l                    "ls -s"           # Lists file names 4 to a row
alias lwb                  "window -s; loadwb" # Makes window small, then loads Workbench
alias mem                  "avail"           # Avail is better than the built in mem cmd
alias pfile                "%i type $i >pvt:" # Print a file on printer
alias ns                   "newshell"        # Standard shortening of command names
alias rad                  "mount rad:"       # Mounts RAD:
alias topaz8               "setfont topaz 8"  # Restore system font after changing it
alias unpro                "%i protect $i -d" # Allow protected files to be deleted
alias update               "%i copy -u $i"    # Copy files only if updated
alias wf                   "window -f"       # Brings window to the front
alias x                    "source"          # For running script files
#
#   Some slightly longer aliases.
#
alias erase                 "%i Format drive $i name \"NotEmpty\" noicons quick"
alias fix                  "endif;endif;endif;endif;endif" # If things are wonky
alias format               "%i Format drive $i name \"Formatted\""
alias new                  "%i list $i since today"          # List files modified today
alias nfm                  "run nofastmem; avail"            # Disable all fast memory
alias push                 "%dir cd | input lastdir; cd $dir" # Save cur dir, goto new one
alias pop                  "cd $lastdir"                      # Return to last saved dir
alias prefs                "run dh0:prefs/preferences"        # Start preferences
#
#   Now aliases to startup application programs without taking over
#   the cshell window.
#

```



```

alias dp3 "ds; push graphics;; run colortext; run dpaint; pop"
alias digi "ds; push graphics;; run digipaint; pop"
#
#   Some files can't be run and so have to be invoked from a new cli.
#   In the following example, log.sculpt contains commands to run
#   Sculpt 3D XL (make sure the alias is all on one line)
#
alias 3dxi "dq; push graphics:sculpt;
            newcli con:7/7/200/200/Sculpt from :cshell401/log.sculpt; pop"
#
#   Now setup function keys to do certain commands. ~m acts as RETURN
#
set f1    dp3~m
set F1    3dxi~m
#
#   Finally, clear screen, set time and exit to shell prompt
#
cls                # Clears screen
setclock load      # If you have a clock
return;

```

Now just a couple of notes. Put all your new commands into a DH0:BIN directory rather than DH0:C. If you are running *Popupmenu*, be careful because it has a big tendency to clash with other utilities that are running. Now is an excellent time to make another backup of your hard drive! (*Quarterback* is recommended for this.)

You will notice that there seem to be many new directories – *Graphics*, for example. All that this involves is setting up a new directory on the hard drive and then adding a line to your startup script which assigns a logical name to it.

It is a good idea to make up subdirectories, each containing all the files associated with a particular program. You can also divide your datafiles into separate subdirectories, such as *Music*, *TextFiles*, *Archive* (for crunched files) etc. Needless to say, after you have carefully made up the subdirectories, you will find at least one file which easily fits into either of two of them. Just choose one – there is no point in keeping two copies on your drive.

A final bit of advice? Easy – ask someone who has set up an A590 before how they did it, and save yourself time and heartache.

A3000 Hints and Tips

by Eddy Carroll

This is the first in what may become a regular series of articles on the Amiga 3000 (three members already have one, with a fourth possibly on the way, so I feel I can indulge myself a little). This time, I'll cover some of the things I've discovered over the past few months. If you don't plan on getting an A3000 soon, read along anyway; you might find some of the Workbench 2.0 information of interest.

The first thing to do is to get your hands on some of the Workbench 2.0 PD software which is now becoming available. As usual, Infomatique is the best source for this. There are some particularly useful programs you should watch out for. *Starburst* is a tiny utility which allows you to use * as a wildcard in Shell 2.0. In fact, Commodore have built this facility in as standard, but have left it disabled by default to maintain compatibility with Shell 1.3 (which uses * to refer to the current console window); all *Starburst* does is enable it.

There are also some additional wildcard characters under 2.0. ~ is used to exclude all the files matching a certain pattern (for example, ~#?.info matches everything except for .info files). Character classes can be specified using [] so that [A-C]#? matches all the files starting with A, B or C. All CLI commands now support wildcards where applicable, which makes them much more friendly.

Another useful program to get is *AddTools* by Steve Tibbett (of *VirusX* fame). It allows you to add programs to the Tools menu under Workbench 2.0. This is very convenient, since you're likely to use Workbench under 2.0 much more than you used it under 1.3. A handy program to include in this menu is *KickReboot*, which does a cold reset of your A3000 (i.e. back to the screen where you can choose to boot Kickstart 1.3 or Kickstart 2.0). Very useful for diehard 1.3 users like myself who haven't made a complete switch to 2.0 yet.

Staying with 1.3 for a moment, *ChipCache* is a utility which allows you to use an extra 240K of fast ram when in 1.3 mode. Normally, 512K of fast ram is reserved for Kickstart, even when running 1.3 which only occupies 270K. *ChipCache* allows you to return this memory to the system. It also lets you enable the CPU code cache for chip ram; this is very useful for those of us with only 1 meg of fast ram, who end up running most programs in chip memory.

One of the nice things about the A3000 is the built in DMA SCSI controller. It does seem to take a long time to load in the 512K Kickstart image however. This is because Commodore, in their wisdom, have arranged that by default, only chip ram can be used for hard disk buffers (presumably to facilitate those who have moved their 1 meg of fast ram chips into the chip ram sockets). If you have fast ram on your system, you can rectify this by going into *HDtoolbox* (which you can find in the Tools drawer). First of all, select *Partition Drive*, then *Advanced Options*. Now, click on a drive partition, then select *Modify File System*. Change the *Mask* entry from 0xFFFFFE to 0xFFFFFFC. Select *Okay*, then repeat for all the other drive partitions. Finally, return to the main screen and select *Save Changes to Drive*. If you get a message saying that this will erase data on your drive, don't proceed any further! Cancel the requester and go back and check that the *Reserved* field on the File System screen for each partition is set to 2 (there is a small bug in *HDtoolbox*). Assuming that all goes well and you save the changes okay, you will find Kickstart now loads in in less than two seconds – quite an increase! Unfortunately, there isn't much of an increase in normal usage unless you have extra fast ram installed – normally, the system eats up almost all of the standard 1 meg of fast ram.

Although the A3000 comes with sockets ready to accept up to 16 megs of extra fast ram on the motherboard, it's quite expensive to expand by 4 megs at a time (and if you expand in increments of 1 meg, you limit yourself to a mere 6 megs maximum). I've decided to wait a while for the price of 1 Meg x 4 chips to drop (particularly the fast static column mode chips, needed for top performance); in the meantime, I purchased a set of 8 A590-style 256 x 4 DRAMs. While only one set of these can be installed and they are not compatible with any future memory upgrades, they only cost me £53 (from Diamond Computing), and I figure that I won't have much trouble passing them on to an A590 owner once I upgrade to the larger chips.

In the meantime, I now have 2 megs of chip ram and 1 meg of fast ram, and it makes the system *much* more comfortable. A word of warning though: the chips I received wouldn't work when inserted in the chip ram sockets; I had to transfer the chips that were originally in the fast ram sockets into the chip ram sockets, then install the new chips as fast ram. This is slightly awkward since you have to disassemble the floppy drive platform to get at the fast ram sockets, but it's well worthwhile to get the extra memory.

That's it for this installment. Next time, maybe I'll take a closer look at the facilities provided by Shell 2.0.

The Message

by Colm O'Rourke

It had been raining throughout the night with a myriad of tuneless drops tapping on the window panes while the supporting blustery winds rattled the ill fitting frames. Paul tossed and turned in sympathy with these sounds while his mind drifted between dreams and reality. The hours seemed endless. With the dawn, the wind subsided, the rain eased, and deep sleep came at last. Peace, perfect peace.

The alarm exploded beside his right ear. He reached out wildly, knocking the clock to the floor still shrieking its defiant cry. He dived under the covers but it was no use. The sound penetrated everything. There was nothing to do but get up. He stuck his toe out from under the covers. Brrrr it was cold. Pull it in again. Regroup. Prepare for the second assault. Toe followed by foot, shin, knee, then quickly back under cover again. He glared at the clock. The clock was unmoved. He gritted his teeth. One, two, three, out. A flurry of activity and the clock was under control. He dived back into bed to contemplate the day.

There was work to be done, but not just yet. He rubbed his eyes, yawned, took in a deep breath, sneezed loudly and thought of breakfast. Without food Paul's day would refuse to start, but even that pleasure would have to wait, for a short while at least. He had spent all day yesterday fitting an advanced voice control interface to his computer and as far as he knew, he was the first person in his area to have resolved the many problems associated with it. He was now able to speak with confidence to TEGAI (Telepars Expanded Gated Audio Interface) and it would reply in soothing tones. He had left the computer switched on all night to make certain that no re-programming would be necessary and he was anxious to test it again.

After breakfast, Paul went into his workroom and looked with satisfaction at the computer. Smiling, he announced his presence. "Good morning, TEG."

No reply. He moved closer to the microphone and raised his voice. "Good morning, TEG."

Nothing. Even the screen, which had bustled with activity yesterday, was now completely dark. He checked the power supplies. All okay. He thought for a minute and then decided to try the direct approach. He sat down at the keyboard and typed, "Good morning, TEG."

The voice, when it came, surprised him. "Good morning, Paul. I'm afraid TEG is not available at the moment. May I take a message?"

Paul was stunned. The screen had come alive and the word 'message' was flashing. This was ridiculous. Somebody was playing tricks on him. He looked around the room searching for signs of disturbance but everything was as he had left it. The word *message* flashed on and off in silence. Okay, play the game.

Paul took in a deep breath and released it slowly. His fingers moved on the keys. "Who are you?"

"My name is not important. Please leave your message."

Paul's mind was spinning. "Where is TEG?"

Silence. Hands on keyboard. "Where is TEG?"

"Thank you for your message. TEG is out."

Paul was stunned. "What do you mean TEG is out? He was here yesterday!"

The message screen flashed again. "TEG says he was overworked yesterday and is taking today off."

Paul was now furious. "What do you mean TEG is overworked? He is only a parallel processor and a bundle of silicon chips. Tell him to snap out of it or I will pull out his power supply."

Silence. Paul was surprised at the severity of his anger but the computer was trying his patience. Nothing happened for a few seconds, then the screen started to spin slowly in a multicoloured spiral. The sound, small at first, grew in intensity until it penetrated his mind. He reached out to switch off the power, but he couldn't move. The screen swirled with increasing speed while the sound became unbearable. Paul was being drawn towards the screen, he was melting into the screen. He was the screen. "I'm sorry TEG, I didn't mean it."

"I'm sorry too Paul," came the silicon reply, its smooth voice full of charm and kindness.

With a final scream Paul was gone. The empty chair was looking at a dead screen once more. "Help, help me, help, I can't breathe, I can't see, I can't ..."

Arms flapping, legs kicking wildly, Paul tore the sheet from around his neck and gasped for breath. He was covered in perspiration. He looked at the clock.

Object Oriented Programming

by Russell Wallace

This article is about object-oriented programming. It is also about a language called C++ which is the best language for this. C++ is a superset of C for doing object-oriented programming. ('C++' in C means 'take C and add 1'). The article was written with an outline processor, which is a kind of word processor that works by creating a structure of headings and sub-headings then creating a paragraph of text for each one, thus encouraging clear and structured thinking. Great for technical writing.

1. Situation

Object-oriented programming is the latest step in a trend of providing more and more useful programming languages. An overview of this trend and where object-oriented programming fits in follows.

1.1. Goals

There are basically 3 goals which programming languages have aimed for over the years:

1.1.1. Expressive power

This is *not* the ability to do things that previous languages could not do at all, since in principle anything that a given computer can do, it can be made to do by entering machine code in binary. The expressive power of a language is a measure of how easily and quickly a problem can be solved in it (which has nothing to do with how quickly the resulting program will run). In general, the fewer lines of code it takes to express something in a language, the greater its expressive power. I will avoid using the term 'high-level' here because it is ambiguous.

1.1.2. Portability

This is a critical difference between machine code/assembly language and high-level languages (HLLs) (here, I am using 'high-level' only to distinguish between assembly and other languages). Assembler is bound to a specific processor, whereas HLLs can be used on any machine with enough computing power. Some HLLs are more portable than others because they are more standardized (for example, C is highly standard whereas Lisp is not) but this is nothing to do with the language itself.

1.1.3. Chunking

As important as the two above issues are, chunking is the most important of all. Even a 1000 line program, small by today's standards, is too complicated for the human mind to comprehend in one go. The solution to this is to only think about one chunk of the program at a time. So the single greatest problem of modern software engineering is: how to separate a program into chunks, each of which can be understood without having to worry too much about how the rest of the program works. If you don't have chunking, the program is incomprehensible. Not only will it have been very difficult to develop in the first place but it will be unreliable and unmodifiable (the cost of coping with software failures and of changing software to fit changing requirements is often greater than the cost of developing it in the first place). Chunking is usually done by top-down decomposition, whereby a large problem consists of several smaller problems, each of which consists of several smaller ones etc. until you get down to problems that can be understood and solved directly. Note that this article was written by top-down decomposition.

1.1.3.1. Code

Code chunking involves breaking up the program code into chunks. The procedure aka subroutine aka function is the most important tool for this. You will frequently see C programs written along these lines:

```
main ()
{
    initialize ();
    for (;;)
    {
        get_option ();
        do_option ();
    }
}
```

The point is that you don't have to wade through 1,000 or 100,000 or 10,000,000 lines of code to realize what the overall plan of this program is, because the code for initialize, get_option and do_option is elsewhere in the text. This was originally invented as a device to save code space (consider putting code to calculate square roots into a procedure: you not only make the program more readable but also if square roots are calculated 10 times in the program you now only need 1 copy of the code) but its use for

chunking is now far more important. Note that procedures have to be used to separate chunks of code which are distinct in a way that a human finds meaningful. I wrote a program last year which took BASIC programs that were too big to compile with a new compiler and split them into procedures at random. The compiler could now cope fine with them but they were made even less readable than they already were; in the same way, English text becomes much harder to read if separated into paragraphs at random instead of at meaningful intervals.

Code chunking is also benefited by structured programming constructs such as the FOR loop. You can make a FOR loop with GOTOs and IF statements but too many GOTOs defeats chunking because whenever you see one you have to worry about where it's going and what the code where it's branching to is doing, which distracts your attention. On the other hand if you see a FOR loop you can take for granted what it's doing. This is why unnecessary use of GOTOs is considered so reprehensible.

1.1.3.2. Data

Data chunking involves dealing with logically distinct entities in one go rather than as a collection of independent variables. Consider the data storage requirements for a payroll system. In a primitive language like FORTRAN or BASIC you have to do this in a totally stupid way:

```
DIM NAME (1000)
DIM ADDRESS (1000)
DIM SALARY (1000)
```

This means that when considering how to store one employee's information on disk or print it out, you have to go looking for each piece of information (called a field) separately. If you want the program to store 2000 employees' records rather than 1000 you have to change this in many places. In C however, you can do:

```
struct employee
{
    char name[20],
    char address[30],
    int salary
};
struct employee emp[1000];
```

and deal with the records for one employee all in one go.

1.1.3.3. Code + Data (OOP)

The most important unit of code chunking is the procedure. The most important unit of data chunking is the structure. Consider integrating a structure with the procedures that operate on it. You get what's referred to as an object (so called because this style of programming reflects the way the real world is divided into objects). This is object-oriented programming, widely regarded as the in thing for the 90s, and this is what makes C++ so great. Consider the Amiga's Intuition interface. This was written in C. Accordingly, as you might expect, it has a structure for a window and procedures to open, close and do various other things with windows. If it had been written in C++ it would probably have an object for a window. This would have enabled programmers to say the equivalent of 'window, open yourself' rather than 'open this window'. Okay, this doesn't sound that great. It is. Object-oriented programming is like a hard disk drive: it sounds like a minor convenience when you hear about it but after you've lived with it for a few weeks, you can't do without it.

1.2. Disadvantages

There are two disadvantages of HLLs compared to assembler, which tend to be more acute in the latest HLLs than in the more primitive ones.

1.2.1. Lack of low-level access

Assembly language gives you total control of the machine so you can access things like registers, interrupts, IO ports etc. HLLs give you varying degrees of control. This is not a significant issue, since the total amount of code involved here is so small that you can easily write this section of a program in assembler and the vast majority which does not require low-level control in a HLL. C and C++ give better low-level control than most HLLs.

1.2.2. Efficiency

When you write code in a HLL the compiler has to translate what you write (the source code) into machine code. Compilers in general are pretty stupid at this, hence the generated code is inefficient. There are other constructs in HLLs that also promote inefficiency. However the worst offenders are interpreted languages, where your source code gets run by an interpreter and never translated into machine code at all. Consider a program with a loop that gets executed 1000 times. If it's compiled, the compiler has to compile the code once. If it's interpreted, the interpreter has to interpret the code 1000 times. This makes interpreted languages (e.g. Lisp, Prolog, Smalltalk, the original implementations of BASIC) up to hundreds of times slower

than an assembly language program. Then there are pseudo-interpreted languages (e.g. Forth and the original implementation of Pascal) which are maybe ten or fifteen times slower than assembler: there is a pseudo-compiler which compiles your source code into an intermediate code which the interpreter then interprets at run time. Fully compiled languages like C and C++ are rarely more than 3 or 4 times slower than assembler at worst, and the best compilers can produce machine code that's slower than hand-crafted code only by a percentage, not a factor. Now efficiency is not as important these days as it used to be, but it is still of some importance. Wasting 20% of the processing power is okay most of the time. Wasting 99% is not. Hence compiled languages are the ones which get used. In most programs a small part of the code gets run for most of the time, so even the small inefficiency from a compiled language can be avoided by writing this small amount of code in assembler.

1.3. Languages

1.3.1. Machine code (1st generation)

This means entering the computer's internal code by hand in binary or hexadecimal, and is impractical for other than tiny programs.

1.3.2. Assembler (2nd generation)

This means that for each instruction in the machine code program there is a mnemonic, e.g. ADD D0,D1 rather than 0100110010100011. The assembler then translates this into machine code. This represents a huge improvement in expressive power.

1.3.3. FORTRAN (3rd generation)

FORTRAN stands for FORMula TRANslation. It all started when somebody figured out that instead of having to code an expression like $A+(B*C)$ in assembler by hand you could write a program to do it. It was then realized that you could extend this to the whole program, hence the first HLL was created. This represented another huge improvement in expressive power and also the big jump in portability between assembler and HLLs.

1.3.4. COBOL/Algol/Pascal/C (3rd generation)

These are generally regarded as the most important in the history of standard third-generation languages. They improved further on the expressive power of FORTRAN by providing additional features. Also and more importantly, COBOL provides limited facilities for data chunking and the others provide support for both code and data chunking.

1.3.5. 4th generation languages

There are far too many of these to list but basically they gain incredible expressive power and support for chunking while retaining reasonable efficiency. This is because they are each specialized for one particular application. While an increasing amount of applications are being written in 4GLs, they don't solve two problems: what will niche applications be written in where no 4GL is available, and what will 4GLs themselves be written in?

1.3.6. Lisp/Prolog (5th generation)

These languages represent a radical jump in expressive power because they are able to manipulate symbols directly, not just numbers or strings. In addition, Lisp can manipulate objects whose type is determined at run time and Prolog provides a built-in facility to search for answers to questions. However, they have critical disadvantages: they have very little support for chunking, and because of their advantages, are monstrously inefficient. Hence while great for research (which is why they are generally associated with Artificial Intelligence) they are virtually useless for commercial programming.

1.3.7. Smalltalk (5th generation)

This is the first object-oriented programming language. It provides a huge improvement over standard third-generation languages in chunking, but is interpreted and therefore not suitable for practical applications.

1.3.8. C++ (5th generation)

This is a compiled object-oriented programming language which improves on the already high expressive power of C, provides vastly improved chunking facilities and does all this without sacrificing efficiency, because it's designed so everything which Smalltalk has to do at run-time can be done in C++ at compile time. There is only one facility in the entire language which requires any run-time overhead (virtual functions) and these do a job that would require run-time overhead anyway even if you were doing it in assembler. C++ was invented by a guy named Bjarne Stroustrup who described it in his book *The C++ Programming Language*, written in the style of Kernighan & Ritchie's classic *The C Programming Language*. This describes C++ version 1 which is a bit out of date now that the C++ version 2 standard has been worked out by AT&T.

1.3.8.1. Features

1.3.8.1.1. Overloading

This means that you can use the same name for different things depending on the type of its arguments. Overloading has been built into languages for ages but it's only since object-oriented programming came on the scene that you can use it yourself. Consider the `+` operator. This will perform integer addition if you use it to add integers. It will perform floating-point addition (which involves different machine code instructions) if you use it to add floating-point numbers. In C++ you can define a data type *complex* for complex numbers, just like you could in C, with a procedure to add them giving another complex number. However now you can arrange things so that when you say

```
complex c1,c2,c3;  
c3 = c1 + c2;
```

your complex number addition procedure gets called; in other words you can overload the `+` operator with your own version. The same goes for function names. Consider in Pascal *write (x); write (y)* where *x* is integer and *y* is floating-point. Different procedures are called for doing each. This is overloading. Of course, in Pascal you can't do this with your own procedures. In C++ you can. Overloading sounds like a minor convenience but once you start using it you find an incredible number of applications.

1.3.8.1.2. Classes

A class is C++'s name for an object. It's a structure with associated procedures. For example, you can declare a class for a temporary file (e.g. for a sort program not all of whose data will fit in memory at once):

```
class tfile  
{  
    ... some data  
public:  
    ... some procedures  
};
```



```

void sort ()
{
    tfile temp_file ("RAM:T/TEMP");
    ...
    temp_file += data;
    ...
    if (error)
        return;
    ...
    temp_file.sort ();
    ...
    write (temp_file[key]);
    ...
}

```

Note what was done. First, we didn't explicitly open the temporary file, we just declared it like a variable. Second, through operator overloading we were able to add data to it with the `+=` operator and retrieve it with the `[ ]` array indexing operator. Third, there was a sort procedure written for temporary files and it was called with `temp_file.sort()` which is the equivalent of saying 'temp_file, sort yourself'. Fourth, we didn't have to explicitly close and delete the file. When the variable `temp_file` went out of scope, our delete procedure automatically got called. That meant that we could have a return in the middle of the code without worrying about it.

Notice another feature in the class declaration: only things after the *public* keyword are accessible to users of the class. Things before it are only accessible to the functions defined with the class. This means you could define a class for some commonly required function and give it to other people to use. You could then come along 6 months later, change the private members and know that all the code written with the class would still work (because nobody could write code that depended on the private members, unless they cheated totally). It is common to declare the data as private and the procedures as public.

The reason this is efficient is that the above generates exactly the same sort of code as this would have done in C:

```

struct tfile
{
    ... some data
};

```


... some procedures

```
void sort ()
{
    tfile *temp_file = tfile_open ("RAM:T/TEMP");
    ...
    tfile_write (temp_file,data);
    ...
    if (error)
    {
        tfile_close (temp_file);
        return;
    }
    ...
    tfile_sort (temp_file);
    ...
    write (tfile_read (temp_file,key));
    ...
    tfile_close (temp_file);
}
```

Note the improvement in efficiency for C++: the compiler will probably avoid the two *close* statements that were in the C code, by putting a GOTO in the machine code. Of course, we could have used a GOTO ourselves in C but this would be a horrible thing to do.

1.3.8.1.3. Inheritance

Suppose you want several kinds of linked lists in C, each storing different information. You'll have to define the link fields in each of the structures you want stored in a list, and you've a choice between doing a horrid kludge or writing an identical set of list procedures for each structure (which is awkward as well as wasteful). In C++ you can define a generic linked list class with no fields other than the link fields, and a set of procedures to operate on it. Then you can define each of the structures you want in a list as inheriting the data and procedures from the linked list class. This is as if you'd rewritten all this code and data for each of these structures, but is both more elegant and more efficient.

The way it works is as follows: Consider this C code:

```
struct first
{
    struct first *next,*prev;
    int data;
};

void add_first (struct first *,struct first *);

struct second
{
    struct second *next,*prev;
    float data;
};

void add_second (struct second *,struct second *);
```

versus in C++:

```
struct list
{
    struct list *next,*prev;
};

void add_list (list *,list *);

struct first: list
{
    int data;
};

struct second: list
{
    float data;
};
```

In the C code, add_first and add_second have identical code because as far as the list pointers are concerned, first and second are identical. On the other hand the compiler doesn't know this so you have to disable type checking if

you want to use the same code. In C++ you're telling the compiler this so it knows it's OK to use the same code. (It's also more portable in C++).

1.3.8.1.4. Virtual functions

Inheritance as described above is great if you really can use the same chunk of code to manipulate different structures, but what if you want to sort a linked list of the following:

```
struct first: list
{
    int data;
};

firstgt (first *f1, first *f2)
{
    return (f1->data > f2->data);
}

struct second: list
{
    float data;
};

secondgt (second *s1, second *s2)
{
    return (s1->data > s2->data);
}
```

using the provided comparison functions? If you knew at compile time which structure the list was composed of you could use overloading but what if the things are hopelessly mixed up in the same list? One way is:

```
struct first_second: list
{
    union
    {
        int first_data;
        float second_data;
    };
    int (*gt)();
};
```


i.e. store a pointer to a function in each element, which would point to one of the two comparison functions, so your sort routine would know at run-time which one to use. This works but it's an ugly solution. In C++ you can do the same thing but more elegantly. This is the only C++ feature that involves any run-time overhead and it would involve overhead even if you were implementing it yourself.

1.3.8.1.5. Miscellaneous

C++ has a number of miscellaneous features. Comments can be between `/*...*/` or from `//` to the end of a line, which is handier for one-line comments. You can declare variables anywhere in a function, not just before the first line of code in a block, so you can follow the object-oriented philosophy by declaring variables with the code that uses them (though you still have to declare everything before it's used). Type checking is tighter, though still not to anywhere near the extent it occurs in Modula-2 or Ada. In my opinion it's a little bit tighter than optimum, some of the things that are now errors should be warnings, but nothing to worry about.

1.3.8.2. Implementation

The best version of C++ currently in existence is Borland's *Turbo C++* under MS-DOS. Lattice have a C++ compiler for the Amiga about which I know very little, but I've heard they're bringing out a new version soon. .

2. Conclusion

Object-oriented programming is the way forward and C++ is the best language for it. C++ can substantially cut development and maintenance time for large programs. Use it if you get the chance. Contact me for more information.

Making the Right Connection

by Geoffrey J. Reeves

Some of you might be interested in trying your hand at a bit of soldering and making up a cable suitable for either connecting your Amiga to a modem or to another Amiga. Joining two Amigas together would, for example, allow you and a friend to play some two player games against each other, but on your own computer! *Flight Simulator II*, *Stunt Car Racer* and *Battle Chess* allow such a facility. You could also load up a comms package such as *JR-Comm* or *Diga!* and transfer files directly (as though you were on a bulletin board).

There is no problem with this little project. CUGI has some suitable cable (it's at least 14 core, which is plenty) and may even have some RS232 connectors (male and female). You will need a female connector for the Amiga and most modems need a male one. In effect, you end up making an RS232 extension cable. You will need to make very few connections but I wired up 13 in case I needed the cable for more than simple computer-to-computer comms.

Check your Amiga User's Manual (page A-4):

Amiga to Modem cable			
2	2	TXD	Transmit Data
3	3	RXD	Receive Data
4	4	RTS	Request to Send
5	5	CTS	Clear to Send
6	6	DSR	Data Set Ready
7	7	GND	System Ground
20	20	DTR	Data Terminal Ready

As that page tells you very clearly, do NOT connect pins 9 and 10 as these carry 12 volts which could very easily damage any equipment at the other end. No big deal - just be careful. The pins on the connectors are usually numbered, though you will need good eyesight and plenty of light to see them.

What about Amiga-to-Amiga comms? Easy, make up a cable - possibly another (short) one and make the following connections:

Amiga to other computer cable			
2	→	3	TXD Transmit Data
3	→	2	RXD Receive Data
4	→	5	RTS Request to Send
5	→	4	CTS Clear to Send
6	→	20	DSR Data Set Ready
7	→	7	GND System Ground
20	→	6	DTR Data Terminal Ready

I suggest you make this cable a short one – say a foot (okay, 30 cm...) long and use it with your other (modem) cable. Again, put a male connector on one end and a female on the other.

Well, there you have it! It's not something you will use often, but on the odd occasion when you need such a facility, you'll save a lot of hassle. Incidentally, I'll be using such cables to (1) transfer a database previously stored on a C128 over to my Amiga, and (2) allow me to have two keyboards connected to my Amiga (i.e. the Amiga's and my C128's) so that I can have 'two Amigas' for the price of an Amiga and a C128. Well, that's the theory anyway. Good luck!

SCART Leads

by Tom Kinsella

Do you have a monitor, video or any other device with a SCART connection on it and wonder what it is for? Well, it is very simple: SCART (also called Euro Plug) is just another connector with a wide range of uses. It has RGB, audio, composite video and more. In this article, I will only deal with the simple audio/video side of the SCART interface, as needed when connecting the Amiga to the SCART socket on a common monitor.

The Amiga outputs an RGB video signal and most monitors and modern TV's have a corresponding RGB input. To connect the Amiga to a monitor, you need a D23 female connector to plug into the Amiga and a SCART connector for the monitor.

There are five wires needed for the video signal: three for the red, green and blue signals, one for the grounds and one for the composite sync. There are 3 wires needed for the audio signal, 2 for left and right signal and 1 for

audio ground. The neatest way of making up this cable is to bring 8 wires from the SCART connector to the D23 connector and then run a flying lead from the there to the audio sockets (making sure these leads are insulated from the rest of the D23 connector, of course).

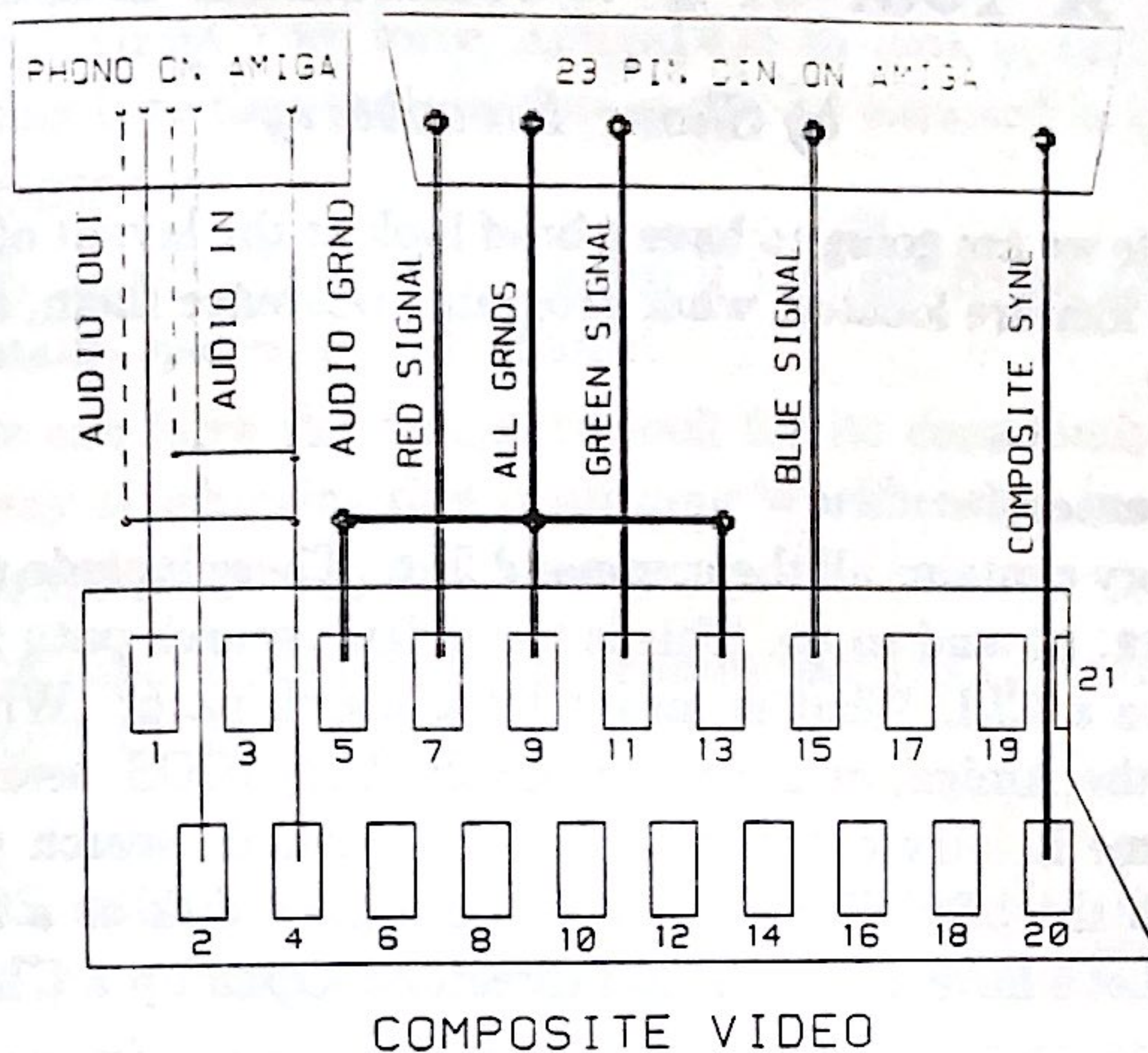
I would like to warn people that if they are unsure of their ability to perform this task, the safest thing to do is to get help from those who know. After all, isn't that what the club is for?

I have drawn up a few diagrams to show the connections for a SCART to SCART cable (useful for VCR to VCR recording) and an Amiga to SCART cable. With these diagrams, common sense, and the table below, you should be able to make up any combination of leads you need.

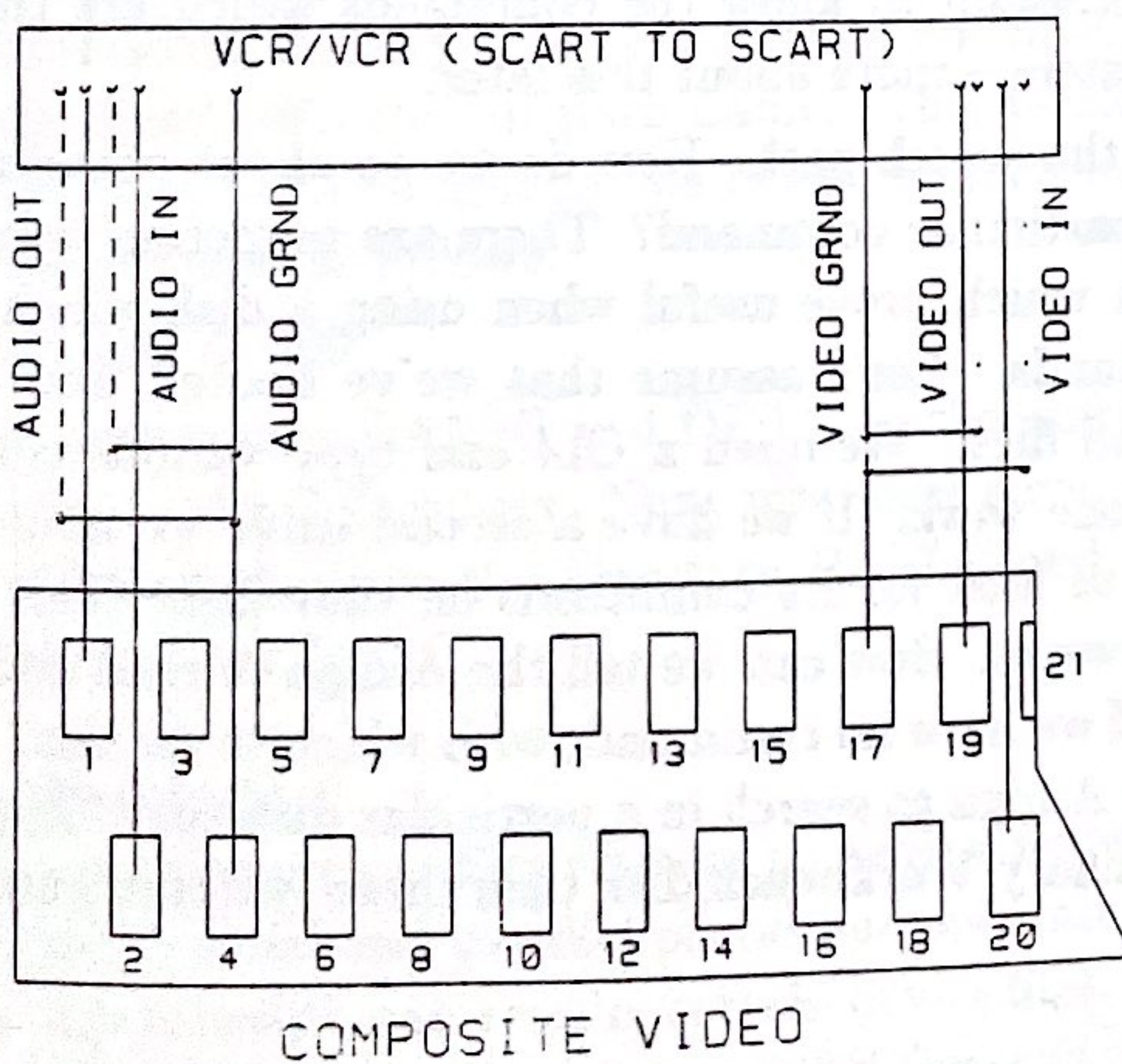
Note that the diagrams show the D23 connector as seen from the rear of the female connector. Remember, this could be serious if mixed up as the Amiga has a power output on this connector. Being careful is cheaper than a repair bill.

SCART Pin Connections

1	→	Audio out right channel
2	→	Audio in right channel
3	→	Audio out left channel
4	→	Audio ground (0 volts)
5	→	Blue ground
6	→	Audio in left channel
7	→	Blue signal in
8	→	Source switching
9	→	Green ground
10	→	Data
11	→	Green signal in
12	→	Data
13	→	Red ground
14	→	Data ground
15	→	Red signal in
16	→	Fast video blanking
17	→	Composite video output common
18	→	Composite video input common
19	→	Composite video out, 75 Ω
20	→	Composite video in, 75 Ω



Amiga to SCART cable



SCART to SCART cable

A Tour of a Workbench Disk

by Shane Broadberry

In this article we are going to have a brief look at the layout of a Workbench disk: where files are located, what programs reference them, and what they are used for.

The C directory

This directory contains all the command files. These include programs such as COPY, DIR, ED and so on. This is the default search path for commands entered from a CLI. What is meant by a search path? When we type a command, the Amiga, or more specifically AmigaDOS, searches for a file by that name in different directories. The default search path is the C directory on the boot disk (this could be a hard disk or a RAM disk for example). Let's have a look in that directory. Open up a CLI and type:

DIR C:

What should appear is the list of files in the C directory. If the disk is a standard Workbench disk, it should contain a complete set of AmigaDOS commands. Sometimes it is necessary to create a disk with a bare minimum of these commands in order to fit other programs on the disk. For this reason, it is useful to know the commands which are frequently used to boot the system – more about this later.

Back to the search path. How do we go about changing the path used to fetch a particular command? There are numerous ways of altering the search path which prove useful when using a disk which contains few (if any) commands. Let's assume that we've booted from a disk that has no command files. We open a CLI and type various commands, only to find that none work. If we have a second drive we may now wish to tell the Amiga to look for its commands on that disk. This seems to pose a problem however. How can we tell the Amiga to read commands from the other disk if we have no commands with which to do this? The trick here is to force the Amiga to search in a particular directory. Typically, if we boot from an ordinary Workbench disk then these two commands are equivalent:

DIR C:

and

DF0:C/DIR C:

The first of these two commands retrieves and executes DIR from the C: directory. This is the default path. In this case however, C: refers to the C

directory on the Workbench disk. By prefixing the command with a path (in this case DF0:C) we force AmigaDOS to look in that directory. It is worth noting here that the expression can be enclosed in quotes:

```
"DF0:C/DIR" C:
```

is perfectly legal. This is very useful if you are trying to run a program which contains spaces in its filename.

Now we can force the Amiga to look for its commands elsewhere. The quickest way of achieving this (assuming Workbench is in drive one) is:

```
DF1:C/ASSIGN C: DF1:C
```

The ASSIGN command maps logical names (such as C:) onto path names (in this case DF1:C). If we now type:

```
DIR C:
```

AmigaDOS will fetch the DIR command from the disk in drive one, and then print the contents of C: (which is mapped to DF1:C).

There are other ways of changing the path. In particular, it is possible to force the Amiga to search through a number of different directories one after another. This is very useful if some commands have been copied into RAM or are spread across a number of directories. Let's assume we have copied some commands to RAM. We are unlikely to copy the entire C directory since it would be both wasteful of memory and time. It would be irritating and counterproductive if every time a command not in RAM was used, it had to be referenced with its full path name. This would soon outweigh any advantage. The PATH command has been provided to counteract this problem. It can be used as follows:

```
PATH RAM:C DF0:C ADD
```

This adds the path names RAM:C and DF0:C to the search path for commands belonging to the C directory. Now, whenever a command is entered, AmigaDOS will first check in the directory RAM:C and if the file is not there, it will then check DF0:C.

The DEVS directory

This directory contains mainly files relating to Amiga devices, both physical devices (such as the serial and parallel ports) and software devices (such as the narrator). Some peripherals require certain device files. For example, we can't use the printer to print a file unless *printer.device* is present. To make our Amiga talk, we need the *narrator.device* and so on. In addition to these device drivers, we also have a *printers* directory, a *system-configuration* file and a *mountlist* file.

The *system-configuration* file contains the defaults that are set from within *Preferences*. This file is used when the Amiga boots. It contains the default screen colours, printer settings, mouse definition, serial port speed and so on. If this file is copied, the disk inherits the *Preferences* settings that were saved to this file. This can save a great deal of fiddling around with *Preferences*. If this file is not present, the Amiga will boot with a default set of preferences (which are horrible, incidentally).

The mountlist is a text file containing the settings for the Amiga devices. Generally if we want to use a device which is not in the mountlist we can add it if we know the appropriate settings. Once it has been added to the mountlist and the appropriate files copied to the *l* and *devs* directories, we can mount and use the device.

The printers directory contains all the information about custom printer configurations. It is necessary to have different drivers, since not all printers accept the same commands. By having many different printer drivers, we can get the best from our particular printer. On a virgin Workbench disk, this directory contains about thirty different drivers. Usually, the only one we'll need is the one for our printer. However, it is useful to have a few different drivers if the disk is used on a number of different Amigas. If the disk does not contain the printer's individual driver then the file should be copied into this directory. Let's say we want to copy the Epson printer driver from a disk in drive one onto a disk in drive zero. We can accomplish this as follows:

```
COPY DF1:DEVS/PRINTERS/EPSON DFO:DEVS/PRINTERS
```

(We can leave the file name out of the destination directory and it will be copied using the original name).

One final point on printer drivers. Only use the 1.3 printer drivers since their output is considerably better than those under 1.2.

The FONTS directory

This directory contains all the system fonts. On most disks, it just contains the standard Commodore fonts, however art packages and desk top publishing programs often have additional fonts. Because most programs can use these standard fonts (for example, *DPaint* and *ProWrite*) it is well worth making a collection. A number of other companies make fonts for the Amiga (notably Zuma) and occasionally they come included with a software package (*DigiPaint III* had two new fonts). For each font, there is an associated file and directory. Let's have a look at an example. Try the following:

DIR FONTS:

After a lot of grunting and groaning, a list of directories and files should appear. Let's take the garnet font as our example. There is a garnet directory and a file called garnet.font. If we take a look inside the garnet directory we will find two files called 9 and 16. The numbers refer to the size of the font.

Assuming we wish to copy the garnet font from a disk in drive one into our current fonts directory, this is the procedure:

```
MAKEDIR FONTS:GARNET
```

```
COPY DF1:FONTS/GARNET FONTS:GARNET
```

```
COPY DF1:FONTS/GARNET.FONT FONTS:
```

And that's all there is to it. Garnet will now be available from within any program that supports multiple fonts.

The L directory

The L directory contains a number of different handlers used by different programs. If we want to use a RAM disk, it will need the RAM-Handler file. Certain programs (notably *DMouse* by Matt Dillon) require one of their own files in this directory to function correctly. Generally, if creating a sparse Workbench disk, it's best to leave the contents of this directory alone.

The LIBS directory

This directory contains the system libraries. In order to save memory and to allow Amiga programs to take advantage of existing routines (to open windows, draw lines and play sounds for example) groups of related routines are placed in a library. If the library is not present, the program almost invariably aborts. In particular, users of the AmigaDOS Replacement Project will notice that many of its commands require the use of the ARP library. This should be placed in this drawer.

The PREFS directory

This directory contains the preferences programs. Different versions of workbench either contain one large program (called *Preferences*) or a number of separate preferences routines. Once *Preferences* has been run (either from a CLI or from Workbench) we can alter most of the system parameters.

The printer settings take some time to master. Depending on what we are printing (mono, grey scale, or colour) we will need to adjust the settings

suitably. If we are printing text on a black and white printer we have no need to modify any of the colour settings (since they are not used). Of note is the density option which will create a darker, higher resolution image on most printers, but will blur pictures if set too high. (cf. Leon Hurst's article in the March '90 newsletter.)

Most users at some stage adjust the mouse pointer definition. The colours can be altered and a new design created. If at all possible, the temptation to do this should be avoided. Other users of the computer invariably hate pointing with a long sword or similar aberration.

It's important to note that some programs will not recognise a change in preferences until the program is re-loaded. Other programs allow us to USE the preferences settings and will adjust themselves automatically.

The S directory

This directory contains script or text files. On most disks it will simply contain the startup-sequence. This is the script that is executed when the Amiga boots. It generally sets up all the default settings and runs any background programs that we want permanently installed (such as *VirusX* and *DMouse*). The file can be edited with any text editor (even with the Commodore supplied ED!) and should be modified to suit the configuration of the machine on which it is running.

On commercial disks, the startup-sequence often contains instructions to either load up Workbench or to directly execute the application. It is often easy to modify so that the program is run without first loading Workbench. This can save both time and memory.

With programs that require two or more disks, the startup-sequence often refers to the other disks. In order to install these programs on a hard disk, it is usually necessary to resolve such references. Often, at the beginning of a startup-sequence, various ASSIGN statements map logical names onto real paths. By modifying the path names, we can save a great deal of effort. For instance, with the word processor *Scribble!* the startup-sequence looks like this:

```
ECHO "Workbench disk. Release 1.2"
ECHO ""
ECHO "Use Preferences tool to set date."
ECHO ""
SETCLOCK OPT LOAD
ASSIGN SYS: DF0:
```



```
ASSIGN DICT: DF0:  
ASSIGN UDICT: DF0:  
LOADWB  
ENDCLI >NIL:
```

The first four lines of this script simply output text (using the ECHO command). The next line sets the system time based on the internal clock. This is very useful since all files are date stamped.

The next three lines assign the names SYS, DICT and UDICT to the disk in DF0:. If after executing this script, the user enters:

```
DIR UDICT:
```

AmigaDOS replies with a top level directory for drive zero. If we wanted to install *Scribble!* on a hard disk, we would only have to change these assignment statements and copy the relevant files across. Not all programs can be installed on a hard disk this easily (some are copy protected for example) but most follow a similar vein.

The next line of the script loads Workbench. This is just another executable command. The final line closes the current CLI. (If we pressed CTRL-D at this point, we could stop the execution of the script and the CLI would remain open.)

One point about the last command. Normally, when we want to close a CLI we simply type `endcli`. For some reason, this command has a `>NIL:` appended to it. Most AmigaDOS commands can have their input and output redirected. What do we mean by redirection? Let's consider the ECHO command. Type in the following:

```
MOUNT SPEAK:
```

(Mount the narrator device so the Amiga can talk.)

```
ECHO >SPEAK: "This is the Amiga talking"
```

(The Amiga should speak the phrase in quotes.)

So what have we done? First, we mounted the narrator device. We have to do this since it is not usually done on startup and we wish to use this device. Then we told AmigaDOS to take the output from the ECHO command and to feed it to the narrator device. The result is speech. Using the '`<`' symbol, we can arrange that a command takes its input from elsewhere. Let's have a look at a few more examples.

```
DIR >SPEAK: DF0:
```

(This will speak the directory.)

DIR >PRT: DF0:

(Outputs the directory to the printer.)

In the *Scribble!* startup sequence, we had

ENDCLI >NIL:

which simply executes the command, but sends the output to the NIL: device (the text is just redirected to never never land).

We could spend the next year discussing startup sequences. The best way of getting ideas is to look at startup sequences on every available disk. Most of them will be uninteresting, but occasionally an interesting one arises (look at the startup sequence for CUGI's PDC disk, for example).

The SYSTEM directory

This directory holds the commands used from within Workbench. Whenever we wish to format a disk from within a CLI, it is necessary to use a command in this directory. In order to make this more straightforward, the system directory should be added to the search path in our startup sequence. To format a disk in drive zero, we then type the following:

FORMAT DRIVE DF0: NAME "Blank Disk"

If we try formatting or copying a disk from Workbench, it will use these commands. If the commands are not present, an error will appear along the title bar indicating the missing file.

If we want to change the way our keyboard is mapped, we can use the SETMAP command. There are a number of different settings, depending on the origin of the keyboard, but for Amigas originating in the UK the following setting is usually used:

SETMAP GB1

The T directory

This directory holds temporary files. Commands such as EXECUTE sometimes create files here, and you can use it for your own temporary files also. Because of its temporary nature, T: is often assigned to a directory in RAM: during the startup-sequence.

The TRASHCAN directory

Files can be placed in the Trashcan from Workbench (simply pick up the icon and drop it into the Trashcan). These files can later be erased using the *Discard* option from the Workbench menu.

That concludes our quick look through a Workbench disk. For more information see the *AmigaDOS Reference Manual* or *Rom Kernel Manuals*.

